Status and Current Research in the Image Understanding Architecture Effort

Charles Weems, Martin Herbordt, Michael Scudder
Computer Science Department
University of Massachusetts
Amherst, MA 01003

James Burrill, Richard Lerner, Thomas Williams
Amerinex Artificial Intelligence Inc.
274 N. Pleasant St.
Amherst, MA 01002

E-Mail Contact: Weems@CS.UMass.EDU

Abstract

The Image Understanding Architecture (IUA) effort is now entering a second phase. The IUA proof-of-concept prototype has been completed and our experience with both the hardware and extensive software simulations are guiding the development of a second generation of the IUA. Furthermore, the initial research-oriented software development environment is currently being replaced by a sophisticated set of application-oriented tools. Thus, the IUA effort is in the process of making the transition from an isolated research project to being in a position of accessibility to the wider community. This article describes the current status of the effort and some of our plans for the future. IUA development is taking place at three sites: the University of Massachusetts at Amherst, Hughes Research Laboratories in Malibu, and Amerinex Artificial Intelligence Inc. The article is thus divided into major sections that describe the efforts taking place at each site.

1. University of Massachusetts

Efforts at the University of Massachusetts have focussed principally in three areas: the design of the second generation IUA hardware, development of advanced programming tools, and algorithm development. The second generation IUA design is nearly complete and, although we expect a few aspects to change, our current view of the architecture is briefly described below. In the area of programming tools, we will give an overview of the multi-associative programming model that we have developed for the low (CAAPP) level of the IUA. We will also discuss some of the issues involved in building a parallel, intermediate-level symbolic representation (ISR) database for the ICAP level of the IUA. We will also summarize an IUA application; that of deriving dense depth maps from known monocular motion.

1.1 Second Generation IUA

For the purpose of comparison, we first summarize the characteristics of the original IUA. The first generation of the IUA is a proof-of-concept prototype containing 4K low-level (CAAPP) processors, 64 intermediate-level (ICAP) processors, and a single high-level (SPA) processor which also serves as the system host [Weems, 1989]. The CAAPP processors are bit-serial, each with 320 bits of on-chip memory and 32K bits of backing-store memory. The ICAP processors are 16-bit TMS320C25 chips, each with 256K bytes of private memory, 256K bytes shared with 64 CAAPP processors, and 128K bytes shared with the SPA. The ICAP processors communicate via a centrally-controlled bit-serial crossbar switch, using their built-in serial communication channels. The SPA is any VME-bus compatible processor, typically a Sun-4.

The Array Control Unit (ACU) for the prototype is a very simple memory buffer for streaming instructions to the array at a high rate. The ACU has no processing or branching capability, and thus all control flow is managed by the host. This arrangement is adequate for its purpose, testing and limited demonstrations of the system, but is not effective for real applications.

Data is loaded into the CAAPP by writing to an image buffer, which is then shifted into all of the processor chips in parallel, via their nearest neighbor mesh. Output from the CAAPP follows the reverse of this process. I/O with the ICAP is performed by the SPA/host, via the dual-ported memory between the ICAP and the SPA/host.

Physically, the IUA prototype consists of 16 12U circuit boards, plus additional boards for control, I/O and communication. The boards are sparsely populated to permit easy diagnosis and rework.

The second generation IUA retains the basic three-level structure of the prototype, but the SPA and host will be separate processors. We expect to use a commercially available multiprocessor board set for the SPA. The host will again interface via a VME extender. The new ACU will be a full-fledged processor, consisting of a microcode engine with a 128-bit instruction word and two separate arithmetic units (one for computation, and one for address arithmetic).

The standard configuration for the second generation IUA will contain 16K low-level processors, 64 intermediate-level processors, and four high-level processors. Physically, the processor array will consist of only 8 12U processor boards, plus some additional boards for control and I/O.

At the CAAPP level, the basic bit-serial architecture will again be used, but a 32-bit corner-turning register increases the on-chip memory to 352 bits per processor. The corner-turning register provides greater flexibility in formatting values that are to be passed to the ICAP. Image I/O with the CAAPP still involves writing to a frame buffer, but the data path to the buffer is now 128 bits wide, permitting a data rate of 160 MB per second. Once the data is in the frame buffer, it appears as merely another segment (HCSM) of backing-store memory (CISM) to the CAAPP processors. Thus, the time to load or store an image is the same as for any other backing store fetch. HCSM provides 4K bytes of storage for each CAAPP processor. CISM has futher been doubled in size to 64K bits (8K bytes) per processor.

The ICAP level has been completely redesigned. It now uses the TMS320C40 32-bit processor, which contains both integer and floating-point units, and operates at up to 50 MFLOPS. Each ICAP processor will have 1 MB of private storage in addition to the ability to access the 2MB of memory it shares with 256 CAAPP processors. ICAP processors are now arranged in groups of four to form a *quadnode* (see Figure 1). Each quadnode has a 4 MB local shared memory which is immediately accessible to the four processors. The local shared memories of all of the quadnodes combine, however, to form a distributed shared memory. Any processor has access to all of the shared memory, although the latency to access a memory outside of the local quadnode will be slightly greater than a local access. In the standard IUA configuration then, there is a 64 MB global shared memory, accessible to all processors. Access to remote segments of the shared memory is via a four by four mesh of buses.

Communication in the ICAP also takes place via a set of message-passing channels. Each processor has six 8-bit channels together with six DMA controllers. Thus, each quadnode has a pool of 24 channels. Of these, 8 form a token ring within the quadnode, 15 are connected directly to all of the other quadnodes, and the remaining channel is

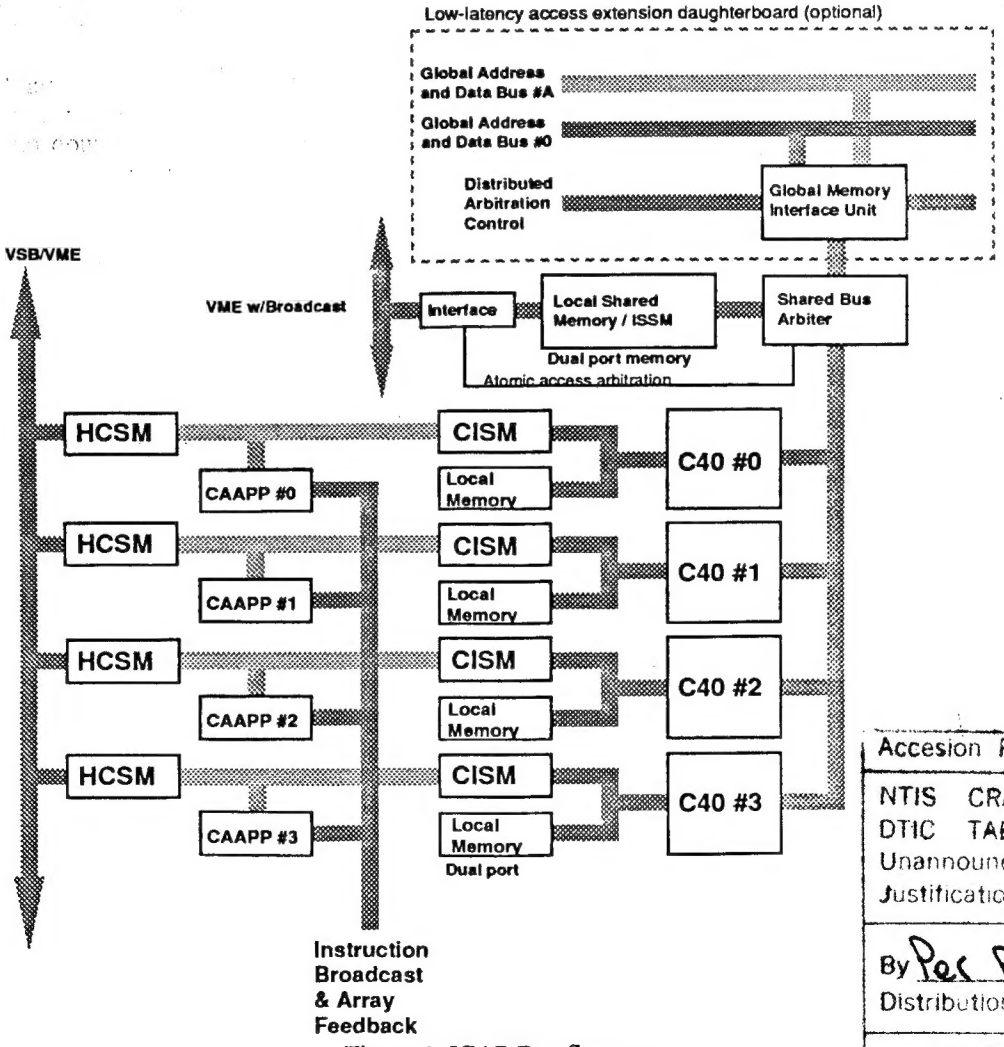

Figure 1. ICAP Bus Structure

brought to an external channel for diagnostics, or custom I/O. Thus, each quadnode is directly connected to all other quadnodes via a DMA channel, as shown in Figure 2.

Note that in Figure 1, a portion of the architecture is labelled as optional. This portion of the system can be omitted in the initial release of the hardware, and then be added later by replacing a daughterboard. In essence, we have designated a minimal subset of the system to reduce the risks in meeting the accelerated development schedule for the second generation. If the optional components are omitted, it is possible to build the entire second generation IUA with

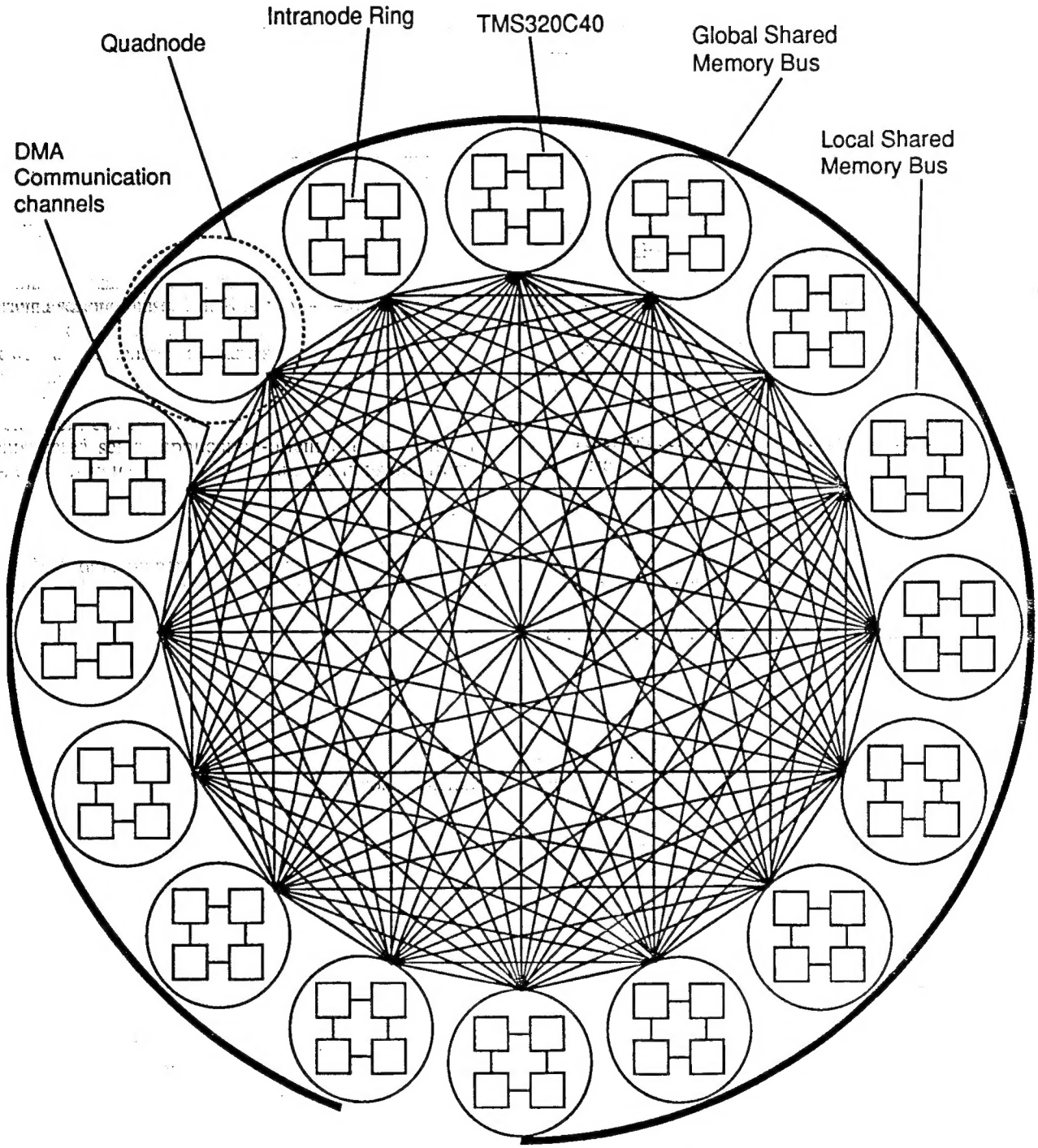

Figure 2. ICAP Communication Structure

off-the-shelf components (except for the custom CAAPP processor chip -- however, that chip is merely a four-times replication of the original CAAPP chip on a single die, so it is also a low-risk component).

1.2 Multi-associative Processing Model

Computation on and among data sets mapped to irregular, non-uniform, aggregates of processing elements (PEs) is an important problem in parallel vision processing, arising in segmentation and in support operations for intermediate-level grouping tasks. The difficulty is that the SIMD processors which map so effectively onto pixel-based processing are restricted in these data dependent computations by the inherent limitations of their control mechanism. Previously, we have used associative processing as a means of applying parallel processing to non-uniform computations [Weems, 1984]. For example, this approach uses global feedback to process individual regions efficiently, but often requires processing to take place on one region at a time. In our current work, we address this problem by introducing an additional level of parallelism, which we call *multi-associativity*, that provides a framework for performing associative computation on independent data sets simultaneously.

A typical vision processing problem to which multi-associativity can be applied is the characterization of regions obtained from a connected components algorithm. Some parameters to be derived may include the number of pixels, boundary length, and mean and median of various spectral quantities. However, since these regions are arbitrarily shaped collections of contiguous processing elements, the communication patterns are also necessarily non-uniform. Although we have developed routing algorithms to collect data using roughly 2d communication steps (where d is the extent of the largest region) [Herbordt, 1990], we would like to take advantage of the constraints provided by this problem to improve that performance.

We first look at how the problem would be solved using traditional associative processing. A typical associative operation is for the controller to broadcast a query to the array, and to receive a response in the form of a count of the PEs with agreeing tag bits. But associative processing, as opposed to the familiar associative memory operations, also enables the conditional generation of symbolic tags based on the values of data, and the use of those tags to constrain further processing. Associative algorithms requiring a number of steps proportional to the number of tag bits have been developed for finding the maximum or minimum value, the mean or median of selected values, and others. Descriptions of these and more complex algorithms can be found in [Foster, 1976; Weems, 1984; and Weems, 1989].

Hardware support in the CAAPP for the global count operation yields performance of approximately 2 micro-seconds; since tag fields are typically 16-20 bits, these associative algorithms complete in roughly 120-200 micro-seconds. Although associative processing enables computations based on PE attributes and relationships to other PEs and events, we are often only processing one region at a time with this approach.

We have developed algorithms for the coterie network [Weems, 1989] to simulate efficiently *within non-uniform aggregates of PEs simultaneously* the associative operations supported directly in hardware by the CAAPP for the entire processor array. Most significantly, we have developed an algorithm to count selected pixels simultaneously in each region in a number of steps proportional to the length of the PE ID (O(logN)). Although this response is not in the micro-second range of the global count, it is significantly faster than previous O(d) algorithms. The consequence is that all existing associative algorithms that were previously run in parallel but region-serially, can now be run region-parallel (on each region simultaneously.) For example, the algorithm to find the mean of some attribute in each region takes O((logN)**2) steps. Although the elapsed time for a single region is significantly longer than the same globally associative algorithm, the gain can still be substantial as often thousands of regions must be processed. We estimate the break even point at between 50 and 100 regions.

Other results we have obtained are new multi-associative algorithms for parallel prefix and convex hull, that is algorithms that perform these computations on aggregates of PEs in parallel and simultaneously. The multi-associative framework also *extends* the traditional associative paradigm by allowing operations on and among aggregates of PEs that are not defined when processing is always performed globally. Two consequences are: the support of divide-and-conquer algorithms within aggregates, and communication among aggregates. The latter operation is especially useful during the merge phase of segmentation algorithms, when characteristics of a region can be transferred to neighboring regions in a single communication step.

1.3 The Object-Oriented Store

A key component of the IUA programming environment is the intermediate-level symbolic (ISR) database [Brolio, 1989; Draper 1989]. The purpose of this database is to store the symbolic representations of extracted image events, groups of events, and instantiated models. As the basis for this new version of the ISR, we are developing a persistent, parallel, object-oriented store.

The store will be object-oriented by Wegner's definition [Wegner 1990], in that it supports objects with classes and inheritance. Objects are the encapsulation of data with the procedures or methods that operate on them. Classes group together objects with a common template. Inheritance uses overloading of functions or operators in a hierarchy of classes to express similarity among related classes.

The store's persistence is derived from the fact that objects may outlive the process that creates them. Thus, programmers are not required to translate between flat file-system storage and structured, encapsulated, in-memory storage.

The parallelism stems from a novel style of parallel programming, known as *POATS*, for Parallel Operators Applied To Structures. POATS provides the speed-up associated with parallel execution but with less programmer effort than traditional MIMD methods. It allows the programmer to concentrate on the data's structure and operations on it, instead of on the coordination of multiple processors. The programmer uses a comprehensive set of data structures and operators to specify transformations of data. POATS is a similar but higher level approach to that of the concurrent aggregates proposed in [Chien 1991].

The operators in POATS can apply functions to data structures in parallel. These functions can themselves be composed of parallel operators so that nested data structures can be handled. Predefined or programmer-created patterns are used to specify dependencies among elements of a data structure. Extensions to the compiler and run-time system of the host language(s) determine the mapping of existing processors to the data structures, and how to coordinate the processors.

POATS combines elements of data parallelism with MIMD processing to permit more flexibility in the manipulation of complex data structures. The POATS model may be thought of as a form of Single Program Multiple Data (SPMD) parallelism, except that it does not necessarily enforce synchronization at the frequent intervals that are typical of SPMD programs. For example, several POATS operations might be active at once, allowing greater utilization of processing resources.

The object-oriented store will eventually become the core of the parallel ISR database. In addition to the capabilities of the object-oriented store, the database will provide meta-data descriptions (schemas), indexing structures, a query language, version control, garbage collection, recovery, and perhaps protection. Built-in objects will include images and image sequences, the DARPA Image Understanding Environment set of standard objects, and libraries of reusable procedures.

1.4 Dense Depth Map Application

An SIMD depth from motion algorithm has been implemented for the IUA using the simulator for the IUA prototype. Image correspondences are established through correlation for two temporally separated images. The depth map is computed from the image displacements and approximately known motion parameters. The map is then filtered to eliminate some possibly erroneous isolated depth values.

The algorithm takes roughly 0.53 seconds to compute on the IUA. By comparison, a similar algorithm for correspondence alone takes about 3 minutes on the Connection Machine, about 10 minutes on a Sequent Symmetry multiprocessor (12 Intel 80386 processors), and about 2 hours on a Vaxstation 3100. The majority of the time is spent on the correspondence, which involves searching a 41 x 41 window in the second image of the pair.

Qualitatively, the algorithm appears to give good results, clearly distinguishing the depths of strong features. Quantitatively, the results are accurate to a range of about 50 feet, for a four foot forward motion of the camera (which has a 45 degree field of view), with a 7.7% mean error in the calculated depths.

2. Amerinex Artificial Intelligence

Efforts at Amerinex Artificial Intelligence (AAI) have concentrated in two areas: completing the design and development of a C++ class library for the CAAPP (initially begun at UMass), and building IU-specific application development tools for the intermediate (ICAP) level of the IUA. Unlike the research-oriented software produced by the University, the effort at AAI is intended to create production-quality software support for the IUA.

2.1 IUA Software Philosophy

Here we describe the progress in creating the software environment that will exist on the IUA. The usual goals were placed on the ensuing environment:

- It should be easy to use.

  As an example, the processing elements (PEs) at the CAAPP level are bit-serial devices requiring 17 operations to add two eight-bit values. It should not be necessary for the user to write these 17 operations or to think of the CAAPP as a bit-serial device.

- It should integrate the various levels of the IUA.

  The IUA consists of several levels of processors and multiple communication paths between and within these levels. It should be possible for users to easily integrate these levels in their problem solutions.

- It should be efficient.

  The incentive to use the combined hardware/software environment, due to increased performance and programmer productivity, should be significantly greater than the incentive to use a standard uniprocessor environment.

- The environment should be familiar.

The user should not be required to learn new and esoteric languages. It is difficult enough to utilize the concepts of parallel processing without having to learn entirely new syntax.

As a general philosophy, we have decided to base the software on objects and C++ where ever this is feasible. As a start, and a framework upon which to tie together the requirements, we developed a Class Library for the IUA. We therefore utilize C++ and a set of classes to describe operations performed at the CAAPP level of the IUA. The programs written using the Class Library look like conventional C++ programs, but where expressions such as (a + b) / (c + 5) may refer to data parallel operations on a SIMD array. These user programs actually run on the Array Control Unit (ACU) and communicate with the host processor using messages (requests). The same programs communicate with the ICAP by invoking processes at that level. We intend to provide libraries that implement various communication schemes. Eventually, to obtain greater object code optimization, we will modify the C++ compiler to recognize our classes, but these changes will not affect the language definition.

In the following section we briefly describe the Class Library for the IUA and how programs written with it communicate with the other parts of the IUA. We describe the major process that runs on the ICAP, and explain how other ICAP processes, both predefined and user defined, are created and communicate.

2.2 The C++ Class Library for the IUA

By using C++, we avoid defining a new language and having to validate its syntax and semantics. C++ provides proven mechanisms for programming that can be used to control the additional operations needed for the IUA. We provide these additional operations using classes (or objects) which are defined using standard C++ and object oriented concepts.

The base class is the *plane*, which is understood to be a two-dimensional grid of elements where each element of the grid exists at a single (virtual) processing element (PE) of the CAAPP. We do not use the term array as it already has a defined meaning in C++ for another construct. In contrast to planes, the nominal objects in C++ are referred to as scalars. Standard arithmetic operations may be applied to planes by applying the operation to each element of the grid that comprises the plane. Standard arithmetic operations may be applied between scalars and planes as well by replicating the scalar for each element of the grid.

Just as scalars are distinguished as being of type *int*, *short*, *char*, etc., planes are also distinguished as being of type *IntPlane*, *ShortPlane*, *CharPlane*, etc. These new classes are derived from the plane class and differ in the number of bits used to represent each element in the grid.

Two levels of control must be provided. The first is "should an operation be applied to an entire plane" and the second is "should an operation be applied to a particular element of a plane". The first type of control is provided by the C++ control statements such as *if* and *for*.

The second method of control is provided by the concept of activity. The activity is specified independently for each virtual PE. Activity is embodied in three new classes *Select*, *SelectNot*, and *Everywhere* which allow the activity to be set for each PE using a BitPlane. Activity controls data transfer within the PE and no other operations. That is, an element in a destination plane is modified under an assignment operation if and only if its PE is *active*. A particular activation has a scope in the same way that variables have scope. Activity may be nested, allowing a cumulative winnowing of the set of active elements.

Other operations are provided for planes. These operations are applied as C++ methods to particular planes. Examples include

- *Any*, which returns a scalar 1 if any elements of a BitPlane are 1 and a 0 if no elements of a BitPlane are 1.
- *Count*, which returns a count of the number of elements of a BitPlane which have the value 1.
- *West*, *North*, *East*, and *South* which implement neighbor communication on the grid.
- Generalized routing operations with combining.

In addition to the the above operations that exist on many SIMD mesh parallel processors, the IUA has hardware for allowing operations to be performed in parallel by regions. This hardware embodies what we believe are important capabilities for image understanding applications. There are four switches at every PE that allow four-way connectedness. A PE is connected to its neighbor if its switch, in that direction, is set and its neighbor's switch, in the opposite direction, is set. Once the switches are set, all PEs that are connected define a region called a *coterie*. Information may be broadcast by some PEs on the circuit formed by the switches and sampled by every PE also on the circuit. Thus, if only one PE per coterie places information on the circuit, it can do a one-to-many broadcast of this information to all the other PEs that form the coterie. If more than one PE broadcasts, the message is the logical OR of the multiple messages sent. For a one-bit message, the result is thus equivalent to the Any operation being applied in parallel to all coteries. Coteries are implemented by the classes *CoterieWENS*, *CoterieWE*, and *CoterieNS* as well as by methods applied to planes.

Because the IUA will exist in several geometries that result in different grid sizes for the CAAPP, it must be possible to write the programs based on the plane size and not the IUA

size. It must also be possible to run the same size problem on both large and small instantiations of the IUA with the only difference being the length of time needed for the computation. A single program may contain several plane sizes. Therefore, programs written using the Class Library specify the size of each plane, and the IUA software maps this to the actual machine that is being used.

For example, if an IUA has a 128 x 128 grid of physical PEs at the CAAPP level and the size of a plane is 256 x 256, then the plane must be split into 2 x 2 tiles to fit the actual IUA. A plane size of 256 x 258 would require 2 x 3 tiles and the tiling factor would be 6.

The size of the plane is specified at compile time by automatically converting the specified size to an integral multiple of the size of the IUA. For example, if the plane size is 40 x 40 and the IUA size is 32 x 32 PEs, there will be four tiles and the actual size will be 64 x 64. We make a distinction between the problem and actual sizes. The programmer must consider the actual size for mesh communications with the *West()*, *East()*, *North()*, and *South()* methods using the torus connections of the mesh. Note that the actual size is the size of the virtual processor array and is not necessarily the same as the physical size.

One of the benefits of this class library is that it does not require the use of an IUA. The class library can be implemented on other SIMD architectures with more or less ability to support the operations provided by the library. It has also been implemented on sequential machines. The generality of the Class Library for the IUA allows it to form the basis of a language for specifying a wide range of image understanding algorithms.

Figure 3 is an example function which calculates the integer square root for each element of an IntPlane. Note how similar this function is to one for scalars.

The function shown in Figure 4 implements a simple edge operation in the x-direction, and is an example of neighborhood communication.

The function in Figure 5 forms regions based on connected component equivalence classes and then labels the regions formed using the address of one of the PEs in each region; it is an example illustrating the use of coteries.

2.3 The ICAP and the ISR

Software for the IUA's intermediate (ICAP) level is arranged hierarchically with each layer providing additional functionality or an abstraction of the lower levels.

Figure 6 depicts the hierarchy. This section describes our current designs for each layer. None of the components have been implemented yet.

```
ShortPlane
IntSqrt(IntPlane initial)
{IntPlane guess (initial.Size());
IntPlane last_guess(initial.Size());
IntPlane res     (initial.Size());
BitPlane a       (initial.Size());
int iterations = 18;
int count;

  guess = initial;
  a = (guess != 0);
  count = a.Count();
  {Select active(a);
   while (iterations--) {
     last_guess = guess;
     {Everywhere active;              // Set 0s to 1s
      {Select active(guess == 0);     // because divide will
        guess = 1;                    // be done everywhere
       }
      }
     res = initial / guess;
     res += last_guess;
     guess = res >> 1;
     if (count <= (guess == last_guess).Count()) break;
     }
   }
  return ShortPlane(guess);
}
```

Figure 3. C++ Class Library Example of Calculating the Integer Square Root of Every Pixel in an Image

```
ShortPlane
prewitt_x(UCharPlane image)
 {ShortPlane x(image.Size());
 //  Compute the first derivative in the X axis direction
 //  with a simple edge operator that applies this mask:
 //   -1 -1 -1
 //    0  0  0
 //    1  1  1
   x = image.South() - image.North();
   return x + x.West() + x.East();
}
```

Figure 4. C++ Class Library Example of Neighbor Communication

A user's ICAP program consists of a set of entry points; the ACU causes the intermediate level to begin execution at one of these entry points as part of the execution of the user's ACU program.

Once begun, the ICAP program performs some complex operation which may involve communication with other portions of the IUA, communication among ICAP processors, maintenance of a shared database, servicing interrupts, and starting or interacting with additional threads of control (tasks). The program performs these actions with the help of the software components in the hierarchy.

```
// Segment `equivalence classes' into regions by
// comparing the values of neighboring
// PEs and then label each region.

IntPlane
label_regions(UCharPlane eq_class)
 {Everywhere active;     // Insure that every PE participates.
  BitPlane west   (eq_class.Size());
  BitPlane east   (eq_class.Size());
  BitPlane north  (eq_class.Size());
  BitPlane south  (eq_class.Size());
  BitPlane masters(eq_class.Size());
  IntPlane labels  (eq_class.Size());

  // Determine the switch settings for the coteries
  // Do not wrap regions around the grid edge

  west  = (eq_class == eq_class.West ()) &
          ~eq_class.WestEdge_p();
  north = (eq_class == eq_class.North()) &
          ~eq_class.NorthEdge_p();
  east  = (eq_class == eq_class.East ()) &
          ~eq_class.EastEdge_p();
  south = (eq_class == eq_class.South()) &
          ~eq_class.SouthEdge_p();

  // form the regions
  {CoterieWENS pattern(west,east,north,south);

  // Select the active PE with the highest
  // address in a region.
  masters = (eq_class.Index()).RegionSelectMax();

  // Label each PE with the address of the master PE.
  labels = (eq_class.Index()).RegionBroadcast(masters);
  }
 return labels;
}
```

Figure 5. C++ Class Library Example of a Coterie Operation (Connected Component Labelling)

At the bottom of the hierarchy lies an ICAP processing element. These processing elements are arranged into groups of four (quadnodes) with each quadnode linked to every other quadnode via the communication ports of the processing elements (and by a shared memory structure).

On top of the processing elements we provide basic system runtime support using SPOX, a real-time multi-tasking operating system sold by Spectron Microsystems, Santa Barbara, CA. SPOX is a widely used commercial product for real-time applications on digital signal processors. It is expressly designed to be easily ported to architectures such as the IUA. SPOX provides basic system support functions such as simple preemptive scheduling, software interrupts, efficient I/O, management of multiple memory segments, and other functions, with low overhead. SPOX provides the basic tools with which we build the higher-level abstractions that are appropriate to our programming domain, including custom multi-tasking abstractions, and synchronization and communication constructs.

The Task Management layer is an interface to many of the SPOX features concerning tasks. This layer provides both basic routines to start tasks, change priorities, and check on the status of tasks, and more abstract routines such as those for executing functions as separate tasks (i.e., "background processes"). In addition, this layer provides a framework within which interrupts are defined and attached to events, and the framework in which a user's program defines its entry points. Finally, it defines such abstract objects as mailboxes, monitors, and other objects to manage synchronization.

The Communication layer defines the basic routines for communicating with other parts of the IUA. For communication via the processor ports, it presents a simple message passing interface; there are routines to construct messages, have them sent to one or more destinations, and

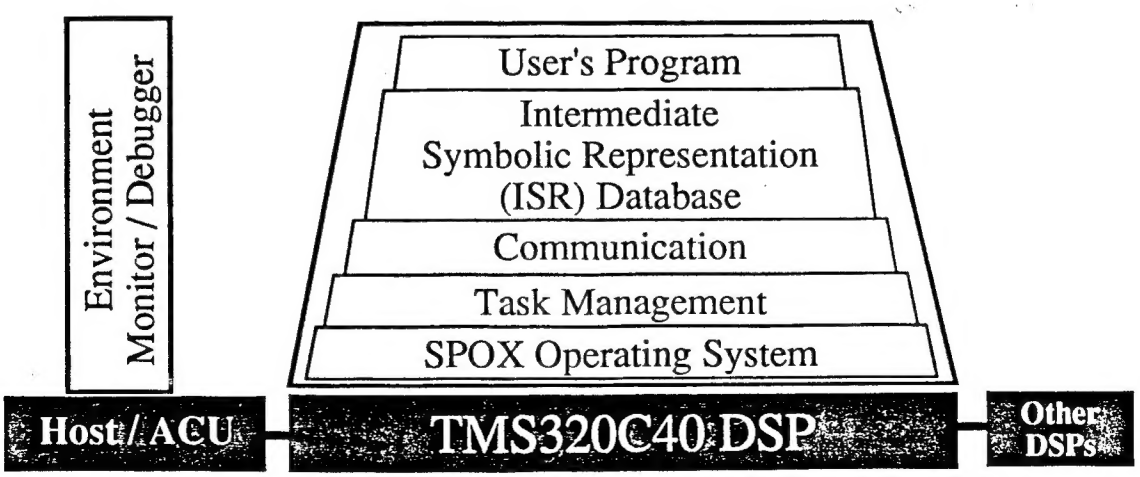

Figure 6. ICAP Software Hierarchy

to receive and dispatch them to the appropriate message handler. For communication via the shared memories, this layer presents both a message passing interface where certain memory locations contain the message queues, and block read/write functions. Both interfaces provide operations that also hide much of the complexity of ICAP--CAAPP communication when running programs with plane sizes larger than the physical CAAPP array. The communication layer also defines such abstract objects as global shared variables.

Although the previous layers are sufficient for intermediate level programming, the interfaces they provide are still fairly primitive. The Intermediate Symbolic Representation (ISR) Database layer provides a higher-level framework within which the ICAP processes can work: that of a shared database of tokens representing image components at various levels of granularity. The usefulness of the ISR database as a framework for image understanding programs has been demonstrated through its use at the University of Massachusetts, where reasearch versions being developed, and as an integral part of KBVision, a commercial research tool for image understanding programming that is a product of Amerinex Artificial Intelligence. The ISR database implemented in this layer is based on that of KBVision. The most significant extensions are those to support a distributed database with limited memory, issues that were not of concern when developing KBVision's ISR. Section 2.3.1 describes the ISR and our extensions.

The final software component in Figure 6, the Programming Environment, provides the tools necessary to run and debug ICAP programs. These tools run on the host machine and interact with the ACU and shared memories. They provide support for loading ICAP programs onto the ICAP processors, loading data into both the shared and local memories, executing initialization routines, and saving memory data after the program has run. Despite the support of the other layers, we expect ICAP programs to be difficult to debug; we therefore require good debugging tools. These tools consist of a portion that runs on the host or ACU and another portion that runs on the ICAP processors. Together they support ICAP program I/O and possibly an interactive debugger. When available, the debugger would allow the setting of breakpoints, program single stepping, and examining memory and registers. The debugger must also be able to handle debugging programs running on multiple ICAP processors simultaneously.

Our layering of the ICAP level software provides a familiar programmer's model (the ISR database) for doing typical processing while providing support for more complex but possibly more efficient management of and communication among the processes running at the ICAP level.

2.3.1 ISR Database

The ISR Database is a repository for information representing abstract image events, such as lines, regions, and edges. It provides tools for defining, storing, retrieving, manipulating, filtering and organizing these events using an interface derived from the ISR database in KBVision [Amerinex, 1991]. Within the database a token describes an image event with a set of features and its spatial location. A tokenset is a group of similar tokens, such as the tokens representing lines extracted by a particular algorithm. A common operation is to find the tokens in a tokenset whose feature values satisfy some criteria. The results of such an operation is a tokensubset, a possibly empty subset of the tokens in the initial tokenset.

On the IUA, a tokenset may be distributed over multiple quadnode IPSMs (memory shared among the ICAP processors in a single quadnode). In a typical operation, ICAP processing begins by populating tokensets with tokens. Since all ICAP processors generate tokens in parallel based on their portion of the image data, the tokens naturally distribute among the quadnodes. This distribution provides a natural partition for parallel computation, but complicates global processing on the tokenset. Our interface supports use of the natural parallelism and hides many of the details of global access.

After populating tokensets, each ICAP typically creates its own tokensubsets for analysis. This operation requires communication among the quadnodes in order to gather token information from the other quadnodes. An ICAP broadcasts a request to every quadnode and constructs the tokensubset with the token information that is returned. Since tokensubsets are unordered, an ICAP can begin processing a tokensubset as soon as data arrives, suspending when it reaches the end of the available data. In the general form of this operation, an ICAP broadcasts to the quadnodes an arbitrary, although previously defined, function and receives a list of the results, which may or may not be tokens. The first form gathers data for processing while the second distributes processing and gathers the results. The general form of the operation may be more efficient than the first depending both on the relative sizes of the input data and the results, and on the data needs of future operations. A process that applies a function to a small number of tokens or that performs multiple operations on a set of tokens may want to gather the tokensubset locally and perform the operations. Broadcasting the function may be more efficient for a process that applies only one function to a large number of tokens, returning a small result (e.g., a summation). We provide an interface that hides many of the details of the tokensubset-specific form. This interface is based on the DynamicList object that implements the more generic form.

2.3.2 Tokensubset Queries

From the user's perspective, tokensubset creation on the IUA is similar to that within KBVision; the user provides a set of tokens and membership criteria and gets a tokensubset in return. However, when using compound criteria, such as those based on multiple token features, the KBVision user creates a tokensubset by successively adding or removing tokens from tokensubsets, while the IUA user creates a

criteria record describing the compound criteria and broadcasts only one request.

Our interface allows tokensubset processing to begin before the entire tokensubset has been received. From the user's perspective, the only change is an additional parameter to all of the tokensubset access functions that specifies whether to block or return when the requested data is not yet available. Non-blocking operations give a "data not available" return code if data is unavailable.

A tokensubset is a simple list of identifiers of tokens in a tokenset. A process uses these identifiers to access the tokens' feature data stored in the tokenset. We provide a form of lazy evaluation of feature values to reduce communication. If a process attempts to access the feature of a token stored in a different quadnode, the system communicates with the remote quadnode to get the information. We then cache this information for future reference. A user's program can prefetch feature data by specifying a list of features when broadcasting its tokensubset criteria. The user's program can also indicate which features are or are not of interest by using some additional directives. Each quadnode records to whom information is sent so that their caches can be updated or invalidated as features change.

The IUA has a limited amount of memory for storing a quadnode's tokens and caching other quadnodes' token data. Long-lived image understanding processing requires some form of garbage collection. Initially, the user's program will be responsible for managing database memory. The user's program running on the ACU is primarily responsible for allocating and freeing tokensets. The ACU allocates tokensets so that they are known to all of the ICAPs. When some phase of processing is complete, the ACU frees the tokensets that are no longer needed. Freeing a tokenset frees the memory associated with the tokens in the tokenset and frees any cached information about tokens in the tokenset. Tokensubsets that are local to an individual ICAP must be freed by that ICAP. Tokensubsets that have been stored as part of a token feature will be deleted when the token's tokenset is deleted. ICAPs also have control over their own quadnode's token data cache through directives indicating when tokens or token features are no longer needed.

Figure 7 shows some of the tokensubset criteria record operations. This and the following examples use C++ syntax although the decision on whether to use C++ or C has not been made yet. When creating a criteria record, the user specifies the tokenset whose tokens are candidates for inclusion within the tokensubset. The user specifies the tokensubset criteria with calls to the *Add* and *Op* operations. The Add operation specifies one criteria and Op specifies the boolean conjunction of multiple criteria. In the arguments to Add, Operation specifies whether matching tokens should be added to or removed from the result, and Test specifies which of a number of tests to use to determine matches. As in KBVision, the test can be one of the following:

- *All*: Match all tokens in the tokenset.
- *Value*: Match tokens whose value in a particular feature is within some bounds.
- *Location*: Match tokens with a location feature that intersects a specified rectangle.
- *Undefined*: Match tokens for which a particular feature value is undefined.
- *NotComputed*: Match tokens for which a particular feature value has not been computed.
- *Criteria*: Match tokens which match a previously defined criteria record.

```
class TssCriteria
 {  ...
  TssCriteria new ( TokenSet );
  void Add ( Operation, Test, TestArgs );
  void Op ( BooleanOp );
  Tokensubset Broadcast ( Destinations, BlockTimeOut );
 };
```

Figure 7. Tokensubset Criteria Operations

The choice of Test determines the arguments that follow. All of the tests, with the exception of All and Criteria, require a feature name and a range of values as the next arguments. All takes no additional arguments and Criteria takes another criteria record as the following argument.

The argument to Op is one of the boolean logic operators: And, Or, or Not. These operate on the "stack" of criteria entered with the Add operation, allowing arbitrary criteria combinations. For example, the following represents a disjunction of two conjunctions:

```
crit.Add(...);
crit.Add(...);
crit.Op(And);
crit.Add(...);
crit.Add(...);
crit.Op(And);
crit.Op(Or);
```

The Broadcast operation sends a completed criteria record to some subset of the quadnodes, and either waits for replies or returns immediately, as specified.

Tokensubset intersection and union are local operations invoked on tokensubsets after they have been received. Thus, these operations cannot be specified in criteria records.

The code in Figure 8 demonstrates the use of tokensubsets and criteria records. This code creates a tokensubset containing the tokens in the tokenset Lines that have start or end points near the start or end points of the line keyLine and have a contrast (indicated as a floating point number) similar to that of keyLine. The first lines of code define the criteria test parameters. The variables dist and conRange

```
ISRFLOAT sminX, smaxX, sminY, smaxY;        \\ area around keyLine's startpoint
ISRFLOAT eminX, emaxX, eminY, emaxY;        \\ area around keyLine's endpoint
ISRFLOAT minCon, maxCon;                    \\ range of acceptable contrast
sminX = keyLine.StartPoint.x - dist;
smaxX = keyLine.StartPoint.x + dist;
sminY = keyLine.StartPoint.y - dist;
smaxY = keyLine.StartPoint.y + dist;
eminX = keyLine.EndPoint.x - dist;
emaxX = keyLine.EndPoint.x + dist;
eminY = keyLine.EndPoint.y - dist;
emaxY = keyLine.EndPoint.y + dist;
minCol = keyLine.Contrast - conRange;
maxCol = keyLine.Contrast + conRange;       \\ Create Tokensubset membership criteria record
TssCriteria crit(Lines);                    \\ criteria record for subset of tokens in Lines
crit.Add(InsertWhen, Location, "StartPoint", sminX, smaxX, sminY, smaxY);
crit.Add(InsertWhen, Location, "StartPoint", eminX, emaxX, eminY, emaxY);
crit.Op (Or);
crit.Add(InsertWhen, Location, "EndPoint", sminX, smaxX, sminY, smaxY);
crit.Op (Or);
crit.Add(InsertWhen, Location, "EndPoint", eminX, emaxX, eminY, emaxY);
crit.Op (Or);
crit.Add(InsertWhen, Value, "Contrast", minCon, maxCon);
crit.Op (And);                              \\ Get Tokensubset
tss = crit.Broadcast(AllQuadNodes, NoBlock);
for(TssIndex = 0; tss.IsIndex(TssIndex, Block); TssIndex )
  {TokenIndex = tss.GetTokenIndex(TssIndex, Block );
xx = Lines.GetFeature (TokenIndex, "FeatureX", Block );
<token computation >
  }
```

Figure 8. Example of Tokensubset Usage

define "near" and "similar to". The line beginning with TssCriteria creates a new criteria record, crit, for specifying match criteria for tokens from the tokenset Lines. The first call to Add specifies a criteria that matches tokens in Lines whose starting point is near keyLine's starting point. The following three lines add criteria that match the other combinations of start and end points. The Or operations indicate that a token matches this criteria if either of its points are near either of keyLine's points. The last Add specifies a criteria that matches tokens with similar contrast. The And operation that follows this Add specifies that matching tokens must both be near and of similar contrast.

After completing the criteria record, we create the tokensubset by broadcasting the criteria to all quadnodes (including the local quadnode), returning without waiting for the replies. The for-loop iterates over the tokens in the tokensubset. IsIndex returns true if the index is within the range of the tokensubset. If some replies are pending and the index is beyond the range of the elements present, this operation will block until an element appears for that index position, or all replies have been received and the index is still out of range. The first line in the body of the loop determines the index within the tokenset Lines of the specified element in the tokensubset. The next line uses the tokenset index to get a particular feature. If the feature is not available locally this operation blocks until it is received. Replacing Block with Timeout(time) would allow these operations to return with status "data not available" if the data was not available within the specified timeout period. NoBlock is synonymous with Timeout(0).

2.3.3 Dynamic Lists

Underlying the tokensubset interface are dynamic lists, a general request/reply mechanism. A dynamic list object assists a program with broadcasting a request to other quadnodes and gathering the replies. The program specifies a function to run on the remote ICAP processor, and a handler to append replies onto a list as they arrive. The system broadcasts this request to the other quadnodes and routes the replies to the appropriate handler. After sending the request, the program may begin processing the list of results as soon as data begins to arrive.

Figure 9 demonstrates the use of dynamic list objects from the caller's perspective. This function finds the token with the greatest contrast. Rather than asking for a tokensubset containing all of the tokens and then invoking FindMaximum on the result, this function asks each quadnode to compute the maximum of its local tokens and reply with this token. The result is a list of tokens, one from each quadnode. The function then invokes FindMaximum on this smaller tokensubset and returns the result. This function also asks for the Contrast and Length features of each quadnode's maximum token.

```
TokenIndex FindMaximumContrast ( tokenset )
  Tokenset tokenset;
{DynamicList dynl ();
dynl.Function ( FindMax );
dynl.AddArgs ( String, tokenset.Name );
        \\ TokensetName
dynl.AddArgs ( String, "Contrast" );
        \\ Feature
dynl.AddReplyFeature ( "Contrast" );
        \\ ReplyFeatures
dynl.AddReplyFeature ( "Length" );
dynl.Handler ( TokensubsetHandler );
        \\ Handler for replies
tss = (Tokensubset) dynl.Broadcast (AllQuadNodes,
        NoBlock);
maxEltTss = tss.FindMaximum(Contrast,Block);
maxEltIndex = tss.GetTokenIndex ( maxEltTss, Block );
return (maxEltIndex );
}
```

Figure 9. Example of Dynamic List Usage

The FindMaximumContrast function first creates a new dynamic list object. It then specifies the function to call and its arguments. It specifies the features what should be contained in any token in the reply, and a function to handle replies. Since the replies will contain token information, the function uses the handler for receiving tokensubset data; the result of the handler will be a tokensubset (i.e., a list of token indices). The function then broadcasts the request, computes the maximum of the tokens in the resulting tokensubset, and returns its tokenset index. The call to Broadcast uses NoBlock so that the following call to FindMaximum can begin its processing as soon as data arrives. This call uses Block so that it waits for all the data to arrive before returning its final result. Using a timeout would let the function return the maximum of the data already received, with a return status of "data not available".

Figure 10 shows the FindMax function that runs on the remote quadnodes as a result of the dynamic list request. The first argument contains information needed when replying to the request. The system provides this argument when it calls the function. The remaining two arguments were provided when initializing the dynamic list object. The function gets the indicated tokenset and uses a criteria record to create a tokensubset containing the tokens for which the feature has a defined value. The argument to Broadcast specifies that only local tokens should go into the tokenset. The function then finds the token with the maximum feature value and creates and sends a reply using the token's tokenset index. The AddData operation automatically inserts the Contrast and Length features into the reply message according to the dynamic list request. The argument to Send can be either Complete or Partial. Complete specifies that this is the last reply from this processor for this request. Partial specifies that more messages will follow. Partial replies allow the originating ICAP to begin processing on partial results without waiting for a long process to complete.

Figure 11 shows the handler for this dynamic list object. The system calls this handler for each reply. The handler returns a dynamic list which is given back to it with the next message. The second argument contains the data in the reply (i.e., the result of the AddData operation in Figure 10). The handler calls cachePartialToken to cache the token feature data in the local database, and appends the tokenset index onto the dynamic list.

```
void
FindMax (requestHeader, tokensetName, feature )
  ...
{
   \\ Get the tokenset based on its name.
    tokenset = GetTokenset (tokensetName );
   \\ Create Tokensubset -- Match all
   \\ tokens with feature defined.
    {
     TssCriteria crit(tokenset);
     crit.Add ( InsertUnless, Undefined, feature );
     tss = crit.Broadcast ( Local );
    }
   \\ Get TokenIndex of token with maximum
   \\ value in feature.
   maxEltTss = tss.FindMaximum ( feature, Block );
   maxEltIndex = tss.GetTokenIndex ( maxEltTss, Block );
   \\ Create reply record.
    {
     DynamicListReply reply( requestHeader );
     reply.AddData ( PartialToken, tokenset, maxEltIndex );
     reply.Send ( Complete );
    }
}
```

Figure 10. Example Dynamic List Function -- Return Token With Maximum Feature Value.

```
DynamicList
TokensubsetHandler (dl, msg );
  DynamicList dl;
  DynamicListMsg msg;
{
   while ( msg ) {
    switch ( msg->type ) {

      case PartialToken:
       cachePartialToken ( msg->partialTokenData );
       dl.Append ( msg->partialTokenData.TokensetIndex );
        break;

      default:
       error ( "unknown message data type" );
       }
    msg++;
    }
  return ( dl );
}
```

Figure 11. Example Dynamic List Reply Handler -- Tokensubset Handler

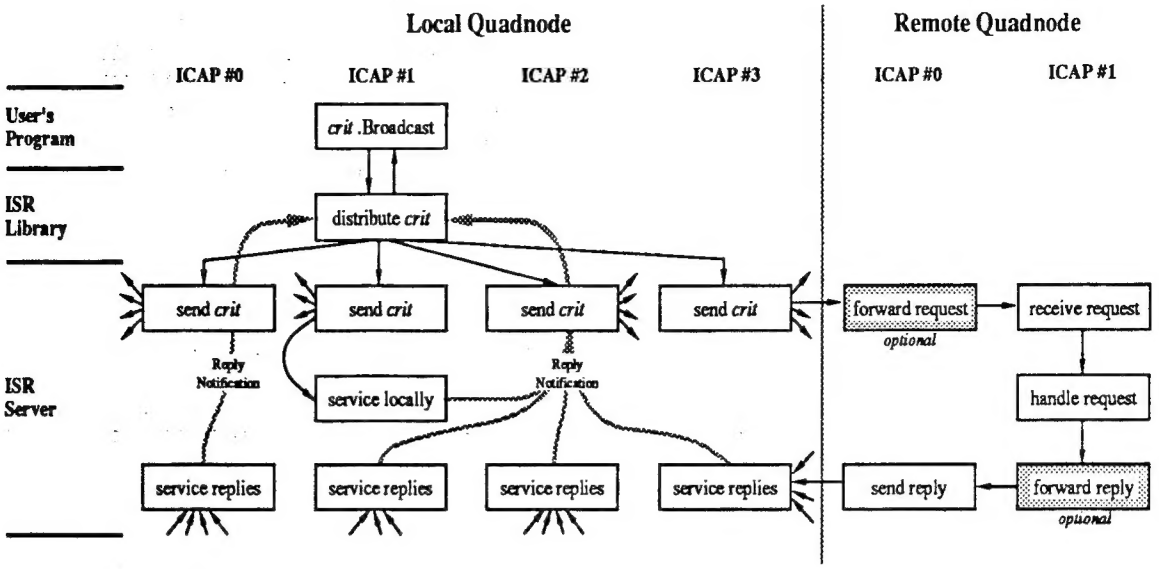

Figure 12. ICAP Communication Example

In this example the resulting dynamic list is a tokensubset that can be accessed with the tokensubset commands discussed above. In other cases, the dynamic list is accessed through its own commands, including indexed access, mapping functions to invoke a function on each element, status functions (e.g., current length and replies pending), and list deletion.

2.3.4 Low-level Communication

Figure 12 depicts the communication resulting from a tokensubset query. The communication lines between quadnodes are distributed among all of the processors in a quadnode. Thus, to send a message to a particular quadnode, the message must first be sent to the processor on the sending quadnode that has the line to the receiving quadnode. Broadcasting messages to all quadnodes requires giving the message to all of the processors on the sending quadnode.

At the top of Figure 12 the user's program invokes the Broadcast operation on a criteria record. This operation is part of the ISR client library and runs as a subroutine call within the user's program. It distributes the criteria record to, and requests service of, each processor on the local quadnode. The ISR server task running on each processor responds to the service request by sending the criteria record to each of its connected quadnodes and preparing to respond to their replies. In addition, one processor is responsible for servicing the criteria request locally as if it had come from a remote processor. As replies arrive, the ISR server task routes the messages to the appropriate handler and notifies any blocked tasks that more data is available. On a remote quadnode, an ISR server task running on the processor with the link to the originating quadnode receives the request and either services it directly or forwards it to another processor in the quadnode. A reply is then sent back to the originating quadnode. If the remote quadnode forwards the request to another processor, that processor must forward the reply to the processor with the link to the originating quadnode, which then sends the reply. The shaded boxes in the figure represent the optional forwarding operations.

All communication among ICAP processors behaves in a manner similar to that of a tokensubset query. Dynamic list requests are nearly identical. For point-to-point communication (from one processor to a particular processor on a remote quadnode) the ISR library routine gives the request to the single processor with the link to the remote quadnode, rather than to all processors; the remainder of the communication is unchanged.

2.4 Inter-level Communication

In this section we describe our plans for other aspects of the software environment being constructed for the IUA. Please note that because the CAAPP is a SIMD array of processing elements whose instructions are generated by the ACU, we treat the ACU and the CAAPP as being one unit in this discussion.

2.4.1 Between the Host and ACU/CAAPP

The Host to IUA connection is that of general purpose computer with a special purpose attached processor. Communication consists of requesting that the IUA, through the ACU, perform some task and return the results of performing that task back to the host. That is, the IUA is an allocated device that performs very complex tasks instantiated in the form of large programs that run on the ACU. Therefore, the host must have a means of initiating tasks on the ACU and then communicating with those tasks as they are running. The host will be executing processes

under some version of Unix. The ACU will be executing tasks under some real-time executive (such as VX Works). Communication will be in the form of messages sent via sockets on the host side and implemented through library functions callable from higher level languages. On the ACU side, this communication will utilize the same semantics but be implemented via a separate library callable from programs written using the Class Library for the IUA.

In a developmental environment, the users will interact with the host and thus with their tasks on the ACU through normal input and print statements. The library for programs running on the ACU will implement the standard C++ stream library functions to provide this interactive facility.

2.4.2 Between the ACU/CAAPP and ICAP

There are three mechanisms for communication between the ACU/CAAPP and the ICAP: via the shared memory layers above and below the ICAP, and via ACU broadcast.

There is a layer of memory, called the CAAPP ICAP Shared Memory (CISM), that resides between the CAAPP and ICAP levels of the IUA and is read/write accessible by both. This memory is used by the ACU/CAAPP for the storage of planes which allows these planes to be available to the ICAP processor and to serve as a means of communication between these levels.

The ACU is capable of broadcasting information to all ICAP processors simultaneously. Programs to be run at the ICAP level are loaded and initiated using this mechanism. Issues of synchronization are handled by library routines available to programs written using the Class Library for the IUA. These routines are based on the broadcast mechanism and the ability to interrupt the ICAP processors.

The ACU also has access to another layer of memory, called the ICAP-SPA Shared Memory (ISSM), accessible by the ICAP and SPA processors. ISSM is addressable from the ACU using a function based upon an ICAP processor's address. It may be used by the ACU to make requests to a particular ICAP processor. The same memory may be used by an ICAP processor to return results or make requests to the ACU (in conjunction with CISM and the CAAPP global feedback mechanism).

2.4.3 Between the ACU/CAAPP and Sensors

The ACU determines when input images are sent to the Host-CAAPP Shared Memory (HCSM) and when images are sent from the HCSM to the outside world. This control is exercised by the user's program through another library that contains routines to control the IDTS (Image Data Transfer System). These routines control the underlying hardware, accessing it via the VME bus. The host also has access to the hardware through the VME bus. But, issues of synchronization with the CAAPP require that only the ACU exercise this control.

2.4.4 Between the Host and ICAP

In normal cases, we do not expect that the host will have a need to directly communicate with the ICAP level of the IUA. While the host has access to some of the same memory that is available to the processors at the ICAP level, issues of synchronization make it unlikely that this facility will be used.

3. Hughes Research Laboratories

Efforts at Hughes Research Laboratories have been directed mostly towards debugging the IUA prototype hardware, designing the ACU for the second generation IUA, the new custom CAAPP chip, and the overall second generation IUA architecture.

The IUA prototype became operational in June of 1991 but, as with most prototype efforts, several problems were encountered. Two of the more serious problems were related to subtle errors in the custom CAAPP chip that were not detected in the circuit simulations. One of the errors involves a control line that passes underneath a portion of the on-chip memory and can cause bits to be lost due to parasitic effects. This error has since been corrected in the second generation CAAPP chip. The second problem involves ground-loop noise due to the spacing of ground pins in the carrier, and will be alleviated by rerouting all ground lines to the inner ring of pins in the second generation chip.

Other problems included resolving interference between Unix and the software initiated memory refresh (refresh is now generated in hardware), compensating for clock skew in the system, and repairing numerous unreliable solder joints.

A preliminary version of the C++ class library, together with the IUA prototype simulator, has been used to develop a missile-tracking related demonstration which has been run successfully on the hardware. In addition, numerous testing and diagnostic routines have been run, and further software is being developed to exhaustively exercise the prototype.

As stated in section 1, a new ACU has been designed that will include a 128-bit horizontal microengine built from AMD 29000 series bit-slice logic. The microengine contains much of the run-time library for the IUA, and is capable of issuing instructions to the CAAPP and ICAP arrays as quickly as they can accept them, and with very little overhead. The instruction issue rate of the ACU is decoupled from the execution rate, and instructions are actually issued asynchronously.

The ACU also contains a "macroengine" consisting of a single-board computer based on a SPARC processor. The macroengine executes the high-level control portion of the user's program and issues instructions to an abstract machine consisting of the microengine and its subroutine library. Thus, a macroengine command might be to perform floating

point division of one plane by another, and the microengine will expand this into the appropriate stream of instructions for the CAAPP array.

Hughes Research Laboratories has also participated in the design of the second generation IUA architecture, developing a separate initial proposal from that of UMass. Ideas from both proposals were combined into the design presented in section 1, and as mentioned there, a few of the details for the design remained to be ironed out, but we expect it to be completed before the end of 1991.

Other efforts at Hughes include the design of the IDTS (which is partially dependent on the specification of the DARPA UGV sensor suite), and advanced packaging for the IUA to further reduce its size while increasing the number of processors.

4. Conclusions

The Image Understanding Architecture is undergoing significant change with the development of the second generation. Both the hardware and the software are being substantially enhanced. We expect the new system to be computationally more powerful than the prototype, and to be much easier to use for vision applications. In particular, the new IUA is being targetted for use in the DARPA UGV program, which will present a wide range of applications challenges, and no doubt lead to further insight into the means for exploiting the potential parallelism in image understanding.

5. Acknowledgements

This work was funded in part by the Defense Advanced Research Projects Agency under contral number DAAL02-91-K-0047, monitored by the U.S. Army Harry Diamond Laboratory, contract numbers DACA76-86-C-0015, and DACA76-89-C-0016, monitored by the U.S. Army Engineer Topographic Laboratory; and contract number F49620-86-C-0041, monitored by the Air Force Office of Scientific Research; and by a Coordinated Experimental Research grant (DCA 8500322) from the National Science Foundation. The authors thank David B. Shu, John Spalding, Dennis Finn, Howard Neely, and J. Gregory Nash at Hughes Research Laboratories for their contributions to the IUA design and development. Thanks also to Rabi Dutta for developing the parallel dense depth map application.

6. Bibliography

[Amerinex 1991] Amerinex Artificial Intelligence, "The KBVision System, Programmer's Reference Manual", Release 2.4, Amerinex Artificial Intelligence, Amherst, MA, June 1991.

[Brolio, 1989] John Brolio, Bruce Draper, J. Ross Beveridge, Allen Hanson, "ISR: A Database for Symbolic Processing in Computer Vision", IEEE Computer, Vol. 22, No. 12, pp. 22-30, December 1989.

[Chien 1991] Andrew A. Chien, Concurrent Aggregates: Using Multiple-Access Data Abstractions to Manage Complexity in Concurrent Programs, OOPS Messenger Vol 2 No 2, April 1991.

[Draper, 1989] Bruce Draper, et al., "The Schema System," International Journal of Computer Vision, Vol 2, No. 3, pp. 209-250.

[Foster, 1976] Caxton C. Foster, Content Addressable Parallel Processors. Van Nostrand Reinhold Company, New York

[Herbordt, 1990] Martin Herbordt, Charles Weems, Jay Corbett, Message Passing Algorithms for a SIMD Torus with Coteries. Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures, pp 11-20. Also in Computer Architecture News, Vol. 19, No. 1, pp. 69-78.

[Weems, 1984] Charles Weems, Image Processing on a Content Addressable Array Parallel Processor. Technical Report 84-14, Department of Computer and Information Science and Ph.D. Dissertation, University of Massachusetts

[Weems, 1989] Charles Weems, Steven Levitan, Allen Hanson, Edward Riseman, J. Gregory Nash, David Shu, The Image Understanding Architecture. International Journal of Computer Vision, Vol. 2, No. 3, pp. 251-282

[Wegner 1990] Peter Wegner, Concepts and Paradigms of Object-Oriented Programming, OOPS Messenger Vol 1 No 1, August 1990.